Bureau d'étude : métro de trottinettes

11 décembre 2006

1 Présentation du bureau d'étude

1.1 Objectifs

Au cours de ce bureau d'étude vous devrez concevoir et implanter sur microcontroleur C167 une loi de commande distribuée. Vous aborderez successivement les aspects relatifs à l'automatique et à l'informatique temps-réel.

...(blabla sur autom)

Concernant la partie informatique les objectifs sont multiples :
- spécifier et concevoir une application temps-réel distribuée ; pour cela nous utiliserons le langage UML car celui-ci a fortement pénétré le milieu industriel ;
- estimer le pire temps d'exécution des tâches et apporter des garanties sur l'ordonancement en appliquant les résultats théoriques vus en cours d'ordonnancement ;
- utilisation d'un noyau temps-réel simple, qui est intermédiaire entre la programmation avec interruption vue en cours de périphérique l'année dernière et entre l'utilisation d'un noyau temps-réel extrêmement complet comme VxWorks vue au cours du bureau d'étude réalisé à SupAéro.

Ce bureau d'étude abouti à la commande d'un système physique réel. Il sera donc l'occasion de vous sensibiliser à certaines caractéristiques physiques difficiles à prendre en compte d'une manière théorique. Nous n'énumérons pas ces difficultés maintenant afin de ne pas en gâcher la surprise ;-) .

Dans la suite de ce document, le travail qui vous est demandé sera encadré, par exemple :

Lisez ce document avec attention dans son intégralité (hormis les annexes).

1.2 Description générale

Les métropolitains actuels sont commandés automatiquement et chaque rame est motrice. Pour des raisons pratiques il faut donc que le moteur de chaque rame fournisse une énergie comparable à celle de ses voisinent.

Architecture physique Au cours de ce bureau d'étude, vous réaliserez un métropolitain réel au détail insignifiant près que les rames seront remplacées par des trottinettes. Chacune

de ces trottinettes (nommé plus loin rames) seront commandées par une microlcontrôleur C167 reliées par un bus CAN de manière à ce qu'ils puissent s'échanger des informations. Un gestionnaire de consigne implanté sur un C167 sera également relié à ce bus CAN.

Au cours de ce bureau d'étude, le métropolitain sera composé de trois rames. Vous devrez cependant prendre en compte le fait que ce nombre pourra être modifié. La figure 1 présente l'architecture matérielle du projet.

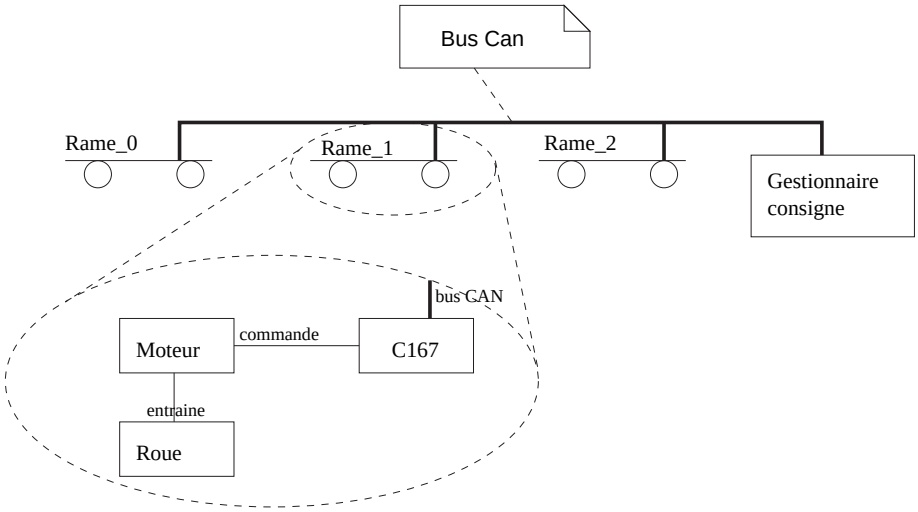

FIG. 1: Architecture du projet

Vue d'ensemble du travail demandé

Le rôle du gestionnaire de consigne sera de commander le métro pour qu'il aille de station en station. Afin de simplifier l'application nous supposerons que le gestionnaire de consigne connaît la station de métropolitain courante et la position de la station suivante. Aucun acteur extérieur à l'application ne commandera donc le système global. Les consignes seront envoyées sur le bus CAN. À chaque station de métro un temps, les rames resteront à l'arrêt pendant un temps fixe. Une fois le temps écoulé, les rames devront aller à la station suivante ce qui revient à asservir les rames en position.

Le démarrage et l'arrêt du métro devra être réalisé « en douceur ». Ainsi, la temps d'accélération au démarrage et de décélération à l'arrêt est fixé à 2 secondes. Pendant ces deux secondes, l'accélération devra être constante.

Le temps d'arrêt entre deux stations est fixé à 3 secondes.

La loi de commande devra prendre en compte le fait que les rames peuvent avoir des réponses différentes, en fonction du chargement de la rame, de l'usure du moteur, etc. et éviter les collisions entre rames.

La figure 2 montre les différentes fonctionnalités (cas d'utilisation) que devra réaliser votre application.

L'électronique chargée de commander et observer l'état du moteur est décrite en section 2. Vous allez maintenant réaliser la partie automatique (cf. section 3) et la partie informatique (cf. section 4)

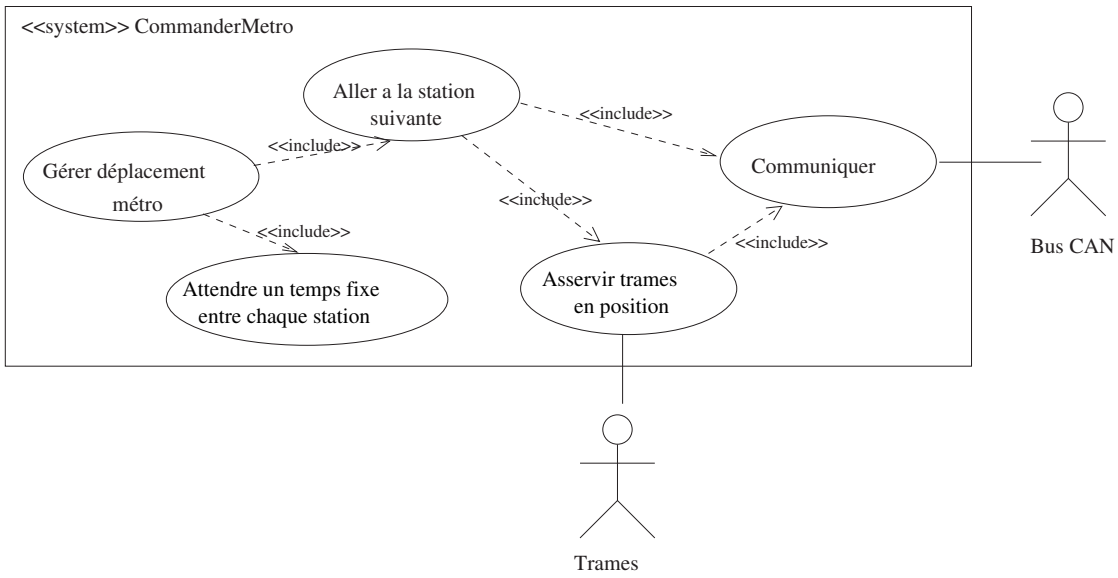

FIG. 2: Cas d'utilisation

2 Description de l'électronique

La partie 2.1 donne une vue d'ensemble du système trame + microcontroleur. La partie 2.2.

2.1 Vue système

Le but de cette partie est de montrer les différents actionneurs et capteurs utilisés pour asservir la trame. La figure 3 montre une vue générale de la commande de la trame par le microcontroleur.

Actionneurs : La commande en `PWM (Pulse Width Modulation)` permet de fixer le couple d'entrée au moteur. Un PWM 50/50 (haut/bas) fournit un couple nul (arrêt), 100/0 fournit la couple maximum vers l'avant et 0/100 fournit le couple maximum vers l'arrière.

Capteurs :

Une génératrice tachymétrique permet de mesurer la **vitesse** courante du moteur. L'acquisition de la vitesse est réalisée au moyen du convertisseur Analogique/Numérique de C167.

La mesure de la **position** est légèrement plus complexe. Deux capteurs optiques sont d'observer les codeurs présents sur la roues (bandes blanches et noires). Des interruptions sont envoyées à l'unité de Capture/Compare du C167 sur chaque changement de couleur.

2.2 Gestion des périphériques

Le but de ce bureau d'étude n'est pas le programmation des périphériques du C167 dont vous êtes déjà des experts renommés grâce à vos qualités intellectuelles hors normes et aux fabuleux enseignants de cette matière. Nous vous fournissons donc une librairie qui permet de gérer ces périphériques.

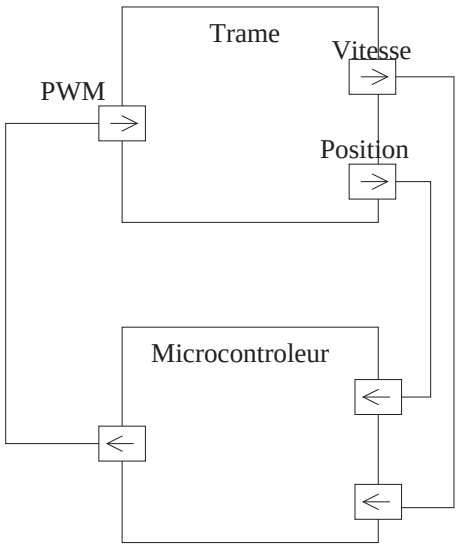

FIG. 3: Vue d'ensemble des actionneurs et des capteurs

2.2.1 Acquisition de la position

Seul l'unité chargée de calculer le position fonctionne sur interruptions. Il faudra donc que vous programmiez le niveau de l'interruption.

Fonctions associées :

- `void Config_Capture(void) ;` : fonction d'initialisation ; de l'unité de capture ;
- `void Init_Position(float Position_Initiale) ;` : fonction d'initialisation de la position ;
- `float Lire_Position(void) ;` : donne la position en mètres. représente la vitesse en $m.s^{-1}$.

2.2.2 Acquisition de la vitesse

L'unité chargée d'acquérir la vitesse consiste à lancer la conversion analogique numérique lorsque la valeur est requise par la tâche d'asservissement.

Fonctions associées :

- `void Config_ADConverter(void) ;` : fonction d'initialisation ;
- `float Lire_Vitesse() ;` revoie un flottant entre -3.3 et +3.3 qui représente la vitesse en $m.s^{-1}$.

2.2.3 Commande de couple

Le couple envoyé au moteur est commandé par un signal PWM.

- `void Config_PWM(void) ;` : fonction d'initialisation ;
- `void Fixe_vitesse(float commande_couple) ;` cette fonction fixe le couple envoyé au moteur ; la variable `commande_couple` est un flottant $\in [-3.3; +3.3]$; par exemple l'arrêt de la trame peut être effectué par l'appel de la fonction `Fixe_vitesse(0) ;` [1].

[1] La nom de cette fonction est trompeur. Il ne faut pas en tenir compte. Nous pensions initialement vous fournir une carte qui régule la vitesse de la trame. Ce que vous devrez faire.

[2]

3 Automatique

4 Informatique

Outils et langages utilisés

Les activités de spécification et de conception tiennent une place de plus en plus importante dans les logiciels notamment dans le développement de logiciels critiques. Le formalisme UML est un formalisme de plus en plus utilisé en industrie (développement de la future rame de métro Parisien ligne 13, logiciel de vol par Thales Avionics de l'A400M, etc.). Afin de spécifier et concevoir notre application nous adopterons donc ce formalisme. [3] Vu le nombre restreint de TP à notre disposition, nous fournissons une grande partie des différents diagrammes. Les activités de spécification et de conception sont présentées aux parties 4.1 et 4.2.

L'implantation du logiciel est réalisée en langage C. L'environnement de développement est le logiciel Keil μ-vision 3 que vous avez déjà utilisé durant les travaux pratiques de périphériques. Les aspects temps-réel sont gérés avec le noyau RTX-tiny. C'est un OS extrêmement simple et la documentation disponible dans l'environnement de développement suffit donc à sa prise en main. Cette partie ainsi que le travail à réaliser sont décrits en section 4.3.

Processus

Le processus mis en place pour ce bureau d'étude est le processus en V. Vous réaliserez donc les parties spécification, conception et codage. Concernant la validation de la spécification et de la conception UML, ceux-ci seront uniquement réalisés par revues « manuelles ». Notez toutefois que des outils de modélisation UML modernes permettent de simuler les modèles dans un but de validation. Ces derniers sont issus du monde industriel (Tau de Télélogic, Rapshody de I-Logix) ou du monde universitaire (IFx du laboratoire Verimag, Turtle du LAAS, etc.).

Concernant l'implantation du système, nous vous proposons une stratégie d'intégration top-down. Celle-ci consiste à coder l'application de manière itérative et à intégrer les différents résultats. Chaque itération traite un problème spécifique de manière à montrer à moindre frais la faisabilité des différentes composantes du projet (traitement de l'aspect communication par le bus CAN par exemple). Chacune de ces itérations donnera lieu à la réalisation d'un prototype livrable.

Prise en compte des contraintes temps-réel Vous devrez apporter des garanties sur la faisabilité de l'ordonnancement des différentes tâches afin de montrer que ces dernières respectent bien leurs échéances (*deadline* en anglais).

Ce bureau d'étude sera ainsi l'occasion :

– de vous sensibiliser à l'estimation du pire temps d'exécution ;

– d'appliquer des résultats théoriques vus en cours d'ordonnancement.

[2] Merci à Thierry Rocachet pour ces bibliothèques de périphériques.

[3] UML est une méthode de conception utilisant le paradigme objet. L'application ne sera pas implantée avec ce paradigme. Nous verrons cependant comment implanter les concepts objets de base (encapsulation des données par exemple) en langage C.

4.1 Spécification et conception préliminaire

L'activité de spécification est une activité primordiale dans le développement des logiciels. Celle-ci tient une place de plus en plus importante et est indispensable sur les projets de grande envergure. Vu le nombre restreint de TP à notre disposition, nous laisserons de coté un certain nombre d'activité de spécification (comme la description textuelle des cas d'utilisation et la description sous forme de diagramme de séquence de ces cas d'utilisation par exemple).

La spécification consistera ici à fournir la structure de l'application (cf. 4.1.1) et à expliciter les différentes fonctionnalités de l'application (cf. 4.1.2).

4.1.1 Vue structurelle

La figure 4 fournit le diagramme de classe global de l'application.

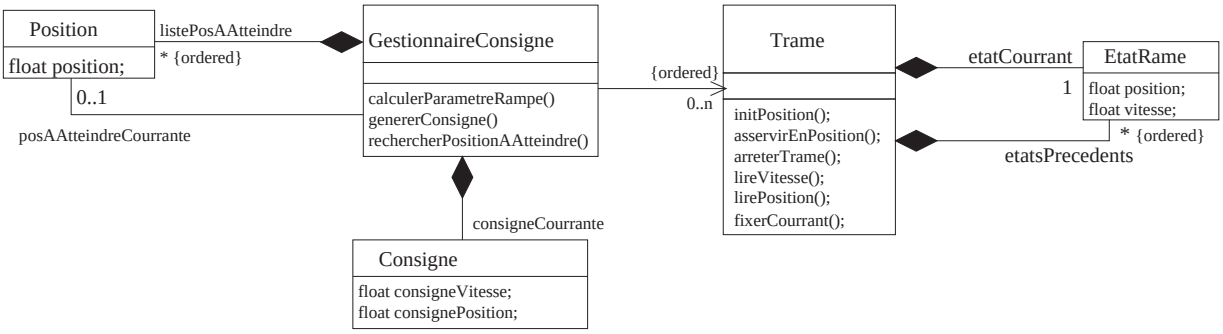

FIG. 4: Diagramme de classes de l'application

Description des classes

• La classe `GestionaireConsigne` contient quatre caractéristiques structurelles :

– `listePosAAtteindre` est la liste des différentes positions auxquelles le métropolitain devra s'arrêter ; ces positions correspondent fait aux positions des différentes stations ;

– `posAAteindre` est la position de la prochaine station à atteindre ;

– `consigneCourrante` est la consigne à envoyer aux différentes rames ; chacune des consignes contient la vitesse et la position du métropolitain voulu.

La classe `gestionaireConsigne` contient deux caractéristiques comportementales (opérations) :

– `calculerParametreRampe()` permet de déterminer les différents paramètre de la rampe de vitesse appliquée au métro entre deux stations[4] ;

– `genererConsigne()` doit envoyer les consignes aux rames, ces consignes dépendront des paramètres de la rampe calculés au préalable ;

– `rechercherPositionAAteindre()` doit mettre à jour l'attribut `posAAteindre`.

L'action `genererConsigne` est spécifiée par le « timing diagram » de la figure 5. Le principe est le suivant. Afin de maîtriser l'accélération et la décélération lors du démarrage et de l'arrêt, il est spécifié que la vitesse maximale et que l'arrêt doivent être atteints en deux secondes. Vous devrez donc être calculer l'accélération `a`, les temps `TpsMontee` et `TpsDescente` transposé en nombre de période d'envoie de la consigne fixée à 10ms. La consigne de position est calculée en intégrant la vitesse. Les différents paramètres devront être tels que la position atteinte à la fin de la rampe soit égale à la position à atteindre.

[4] Ceux-ci seront explicités plus loin.

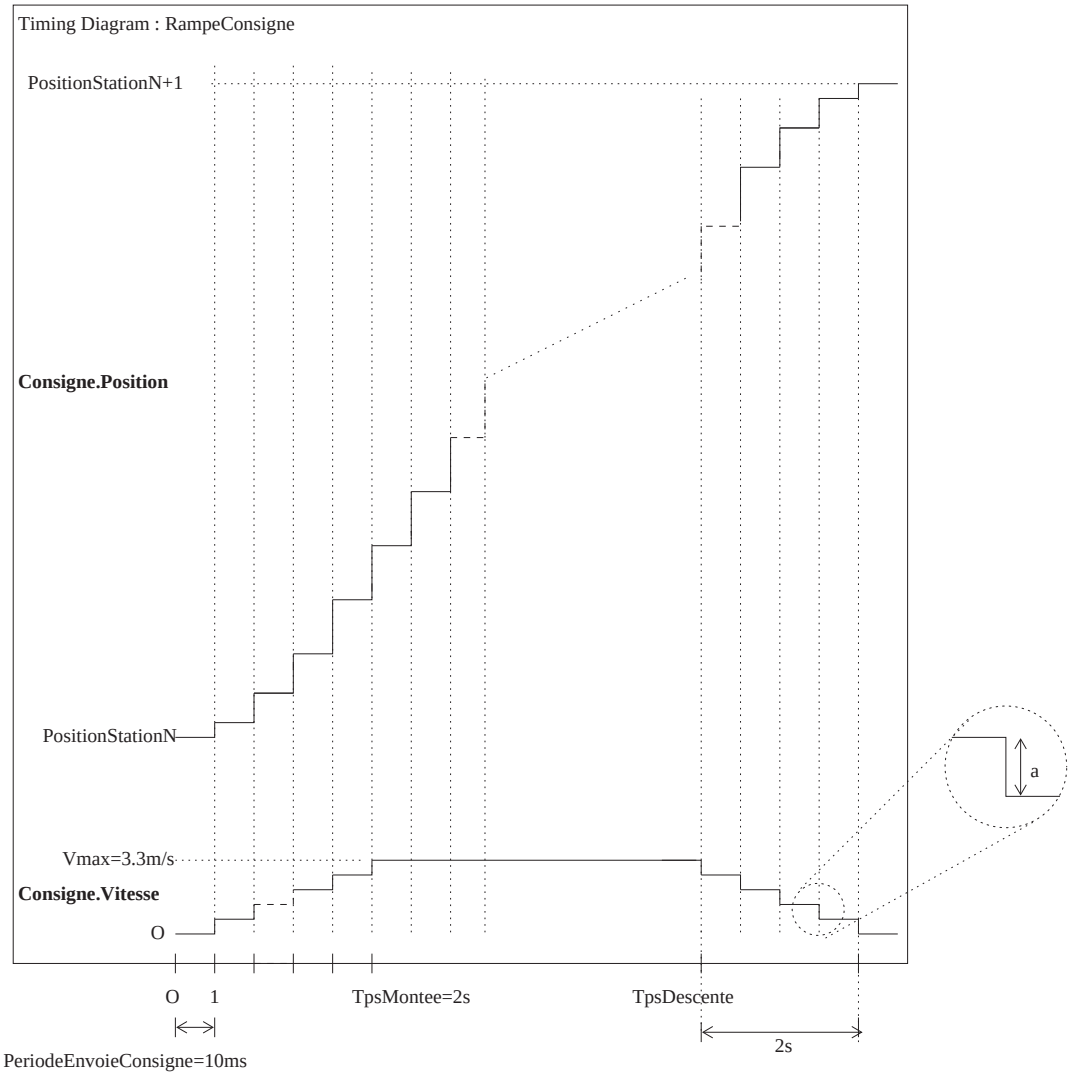

FIG. 5: Spécification de la rampe de position

• La classe `EtatRame` permet de représenter l'état d'une rame à un moment donné. Cet état est déterminé par la vitesse et la position de la rame.

• La classe `Rame` a pour but de gérer une rame. Elle contient deux caractéristiques structurelles :

- `etatCourrant` qui contient l'état courant de la rame ;
- `etatsPrecedents` qui contient un sous-ensemble des états précédents de la rame ; ces états précédents sont utilisés pour l'asservissement de la rame, le nombre des états précédents à mémoriser dépend de la loi de commande.

Voici la description des caractéristiques comportementales de cette classe :

- `initPosition(float Metres)` initialise la position de la rame à une valeur donnée ;
- `lireVitesse(float * Vitesse)` lit la valeur courante de la vitesse de la rame (valeur fournie par le génératrice tachymétrique) ;
- `lirePosition(float * Metre)` lit la valeur courante de la position ;
- `fixerCourrant(float Courant)` fixe le valeur du courant fournie au moteur au moyen

de l'unité de gestion du PWM ; avec `Courant` $\in$ [-3,3 ;+3,3] ;
- `lireConsigne(float * Position)` permet de lire la consigne de position qu'il faut atteindre ;
- `asservirEnPosition()` permet d'asservir en position la rame.

4.1.2 Vue fonctionnelle

Le diagramme d'activités de la figure 6 illustre le séquencement des différentes fonctionnalités effectuées par `GestionnaireConsigne`. [5]

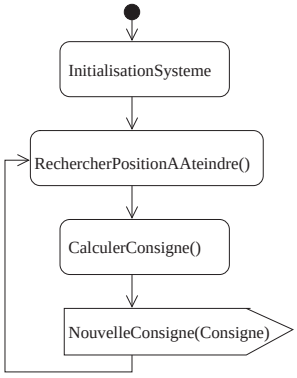

FIG. 6: Fonctionnalités du contrôleur

En vous inspirant de cette représentation, spécifiez les fonctionnalités et leur séquencement de la classe de gestion de la rame. L'annexe B.1 fournit la sémantique (le sens) des différentes constructions d'UML utilisées dans ce diagramme.

4.2 Conception

La conception est réalisée en deux étapes présentée aux section 4.2.1 et 4.2.2.

4.2.1 Conception Générale

La première partie de la conception consiste à assigner les différentes fonctionnalités sur les unités de traitement (tâches). Pour cela vous utiliserez les guides de conception présentés à l'annexe A. Ces guides sont extrèmement utiles pour identifier et justifier la répartition des fonctionnalités sur les tâches. Pour chacun des tâches vous indiquerez, les données d'entrées, les données de sortie, les fonctionnalités réalisées et les différentes synchronisation et contraintes temporelles.

Réalisez un document qui fait la description des différentes tâches comme spécifié précédemment.

[5] Attention, il est fréquent et faux de mélanger fonctionnalités et tâches d'un système temps-réel. En effet, une tâche pourra réaliser plusieurs fonctionnalités (par exemple la tche chargée de l'initialisation du système accomplira sûrement l'initialisation de tous les capteurs et actionneurs) et des fonctionnalités complexes peuvent être implantées en plusieurs tâches (par exemple la fonctionnalité aller à la station suivante).

La deuxième étape de la conception générale est de représenter les différentes tâches de manière graphique. Le but d'un tel diagramme est de spécifier de manière graphique les différentes tâches, leurs interactions, les variables partagées (produites et consommées), les sémaphores utilisés, les différentes périodes. Ce diagramme fait abstraction de la manière dont est réalisée la tâche (l'algorithme interne n'est pas décrit : chaque tâche est vu comme une boîte noire). Par exemple la figure 17 de la partie B.2 est un exemple où le système contient deux tâches.

Représentez les différents tâches identifiées au moyen d'un diagramme d'activité. Pour cela inspirez vous de l'annexe B.

4.2.2 Conception détaillée

Au cours de la conception détaillée nous nous proposons de concevoir le fonctionnement interne des différentes tâches. Encore une fois nous utiliserons le diagramme d'activité.

Afin de préciser en quoi consiste cette étape nous vous présentons un exemple. Supposons qu'au cours de l'étape de conception générale nous ayons décidé de regrouper les différentes fonctionnalités de `GestionnaireConsigne` au sein d'une même tâche. Les figures 7 et 8 décrivent la conception de la tâche `GénérerConsigne`.

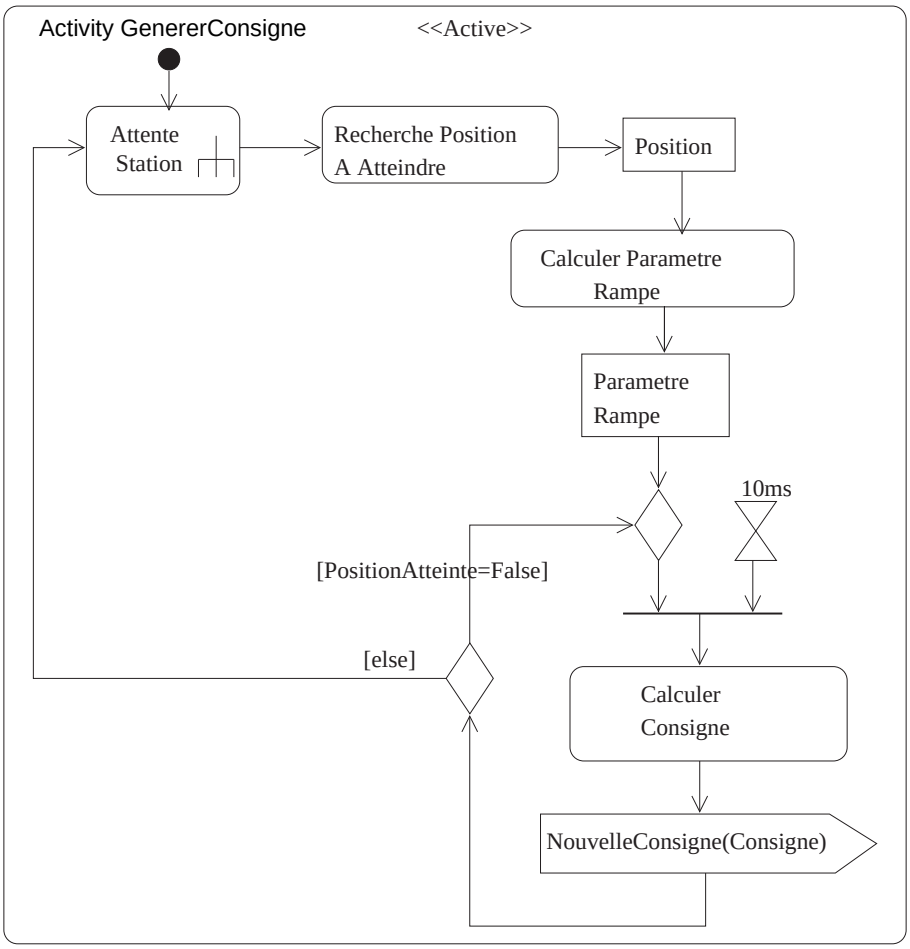

FIG. 7: Conception de la tache GénérerConsigne

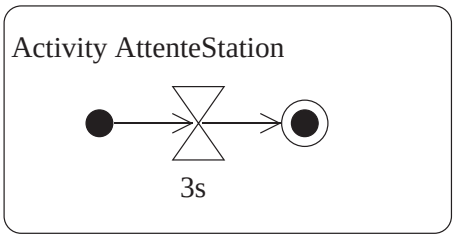

FIG. 8: Conception de l'activité AttenteStation

En vous inspirant de cette représentation, concevez les différentes tâches à effectuer sur chacune des rames. L'annexe B fournit la sémantique (le sens) des différentes constructions d'UML utilisées dans ce diagramme.

4.3 Implantation

La partie d'implantation est découpée en 5 phases. Chaque phase sera validée en simulation puis implantée sur le matériel (hormis phases 3 et 4). Chacune des phase donnera lieu à une livraison. Afin d'augmenter votre productivité vous pouvez réaliser plusieurs phases en parallèle tout en respectant le séquencement de la figure 9.

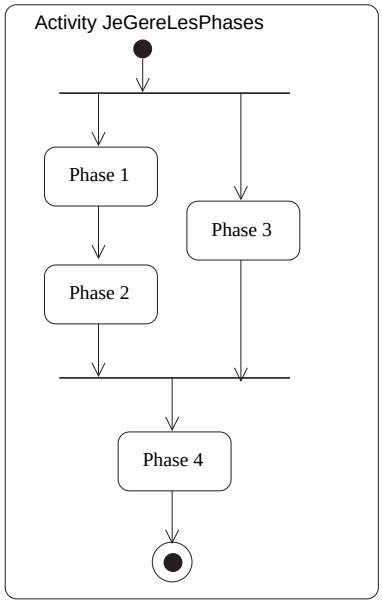

FIG. 9: Je gère les phases dans mon groupe pour pouvoir finir le TP

4.3.1 Phase 1 : Prise en main du noyau temps réel tiny

Fonctionnalités : faire deux tâches qui font clignoter les LED 1 et 2 de la carte MCB167-NET avec des périodes respectives de 0,5s et 1,5s.

Fichier fournis : Fichiers qui permettent de modifier l'état des LED 1 et 2.

4.3.2 Phase 2 : Asservissement en position d'une rame

Fonctionnalités : Faire une tâche d'asservissement en position d'une rame.

Fichiers fournis : Librairie de gestion des périphériques utilisés pour commander le moteur de la rame (PWM) et pour acquérir l'état de la rame (position par unité Capcom, et vitesse par lecture de la génératrice tachymétrique).

Un fichier qui simule le fonctionnement d'une rame (.ini) et qui permet donc de valider la commande en simulation.

4.3.3 Phase 3 : CAN

L'année prochaine le bureau d'étude sera couplé avec les TPs concernant le bus CAN. Aussi cette année, nous ne vous demandons pas de réaliser la conception associée à l'ordonnancement des messages et à l'implantation de la gestion du CAN.

Description des messages :

Message	Émmetteur	Récepteur	Identificateur
Consigne : 2 flottants qui codent pour la position et la vitesse voulue	Gestionnaire Consigne	Toutes les rames	10
État rame 1 : 2 flottants qui codent pour la position et la vitesse réelle	Rame1	Rame 2 et 3	20
État rame 2 : 2 flottants qui codent pour la position et la vitesse réelle	Rame 2	Rame 1 et 3	30
État rame 3 : 2 flottants qui codent pour la position et la vitesse réelle	Rame 3	Rame 1 et 2	40

Connaissant le débit du CAN, et les périodes d'envoie des différents messages vérifiez que l'ordonnancement des messages sur le CAN est toujours possible.

Fonctionnalités :

- initialise les CAN des différentes stations, qui envoie les messages réels.
- lancer une tâche de période 2 secondes sur les 4 C167 qui :
  1. envoie son message sur le réseau, et incrémente les différents flottants ;
  2. lit les messages reçus ;
  3. affiche ces messages (avec un print).

Fichiers Fournis :

Les fichiers d'initialisation et d'utilisation du CAN.

Note : Les identificateurs des messages ne sont pas consécutifs afin de permettre l'ajout de nouveaux messages de priorité intermédiaire.

Les fichiers qui vous sont fournis permettent notamment la manipulation d'un message de priorité forte afin par exemple d'implanter un arrêt d'urgence.

4.3.4 Phase 4 : intégration

Fonctionnalités : Intégrer les résultats des phases 3 et 4 afin réaliser l'application finale.

Fichiers Fournis : RAS

A Guides pour le découpage en tâches

Guide 0 : On définit les tâches selon les choix architecturaux.

Guide 1 : 2 evts sont pris en compte par 1 meme tâche s'il existe une relation séquentielle stricte entre les 2 prises en compte.

Guide 2 : Deux actions sont émises par 1 meme tâche s'il existe une relation séquentielle stricte entre les 2 émissions d'action.

Guide 3 : Un couple (Evti, Actj) est traité par une meme tâche s'il existe une relation séquentielle stricte entre la prise en compte de l'événement et l'emmission.

Guide 4 : Pour chaque tâche définir son rôle : à quoi elle sert dans le systeme.

Guide 5 : Définir les flots de données entre tâches : sens, type des données. À appliquer pour tous les couples de tâches.

Guide 6 : Spécification du type de communication :

- Synchrone si : Tte valeur produite est consommée la valeur suivante ne peut etre communiquée qu'après traitement de la précédente
- Asynchrone par variable partagée si : La consommation ne doit pas ralentir la production Toute valeur produite n'est pas forcément utilisée, seule la dernière donnée est essentielle
- Asynchrone par file si : La consommation ne doit pas ralentir la production Toute valeur produite doit etre utilisée, le consommateur doit pouvoit rattraper son retard.

Guide 7 : Étudier les relation de précédence entre les différentes tâches et exprimer le type de synchronisation.

Guide 8 : Mettre en valeur les ressources partagées (entre tâches) et les actions à faire en exclusion mutuelle.

[6]

B Diagramme d'activité

Le but de cette annexe est de présenter le sens des constructions de base des diagrammes d'activité (cf. section B.1) et de présenter comment il est possible d'exprimer les concepts temps réel au moyen de ces diagrammes (cf. section B.2).

B.1 Constructions de base des diagrammes d'activités

Les diagrammes d'activités d'UML se sont inspirés de la sémantique à jeton des réseau de Petri (bien que les jetons ne soient jamais représentés). Les diagrammes d'activité permettent de représenter le séquencement des différentes actions. Une action peut correspondre à quelque chose de très simple comme l'affection d'un attribut, d'un peu plus abstrait comme l'envoie d'un signal ou de complexe en invoquant une activité.

Séquencement d'action Comme dans les réseaux de Petri, la séquence entre deux actions est décrite de manière séquentielle (cf. figure 10).

Activité Une activité est un ensemble d'actions. Chaque activité peut ou non avoir des paramètres d'entrées et de sorties (cf. figure 11 tiré de la norme d'UML).

Nœuds de contrôle

UML introduit différents nœuds de contrôles permettant de maîtriser le flot de jetons.

La figure 12 en donne une représentation graphique.

[6] Merci à Mr Gilles Motet pour ces guides fabuleux.

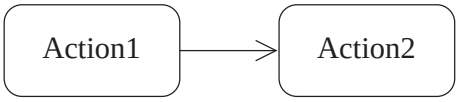

FIG. 10: Séquencement d'actions

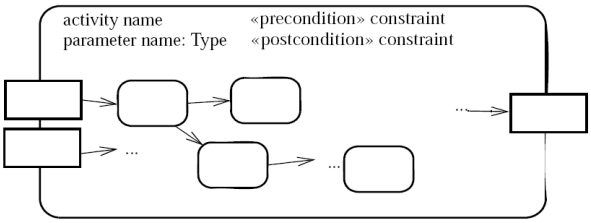

FIG. 11: Représentation d'activités

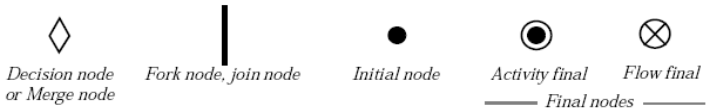

FIG. 12: Nœuds de contrôles

La figure 13 donne un exemple d'utilisation. La barre modélise une duplication de jetons (*fork*) ou une synchronisation de jetons (*join*) (selon le sens d'utilisation), le losange exprime un choix dans la distribution des jetons (*decision*) ou le rassemblement de deux chemins sans synchronisation (*Merge*). Ainsi, sur la figure 13 si `Condition=true` les actions `B` et `C` seront accomplies, et les actions `A` et `C` sinon.

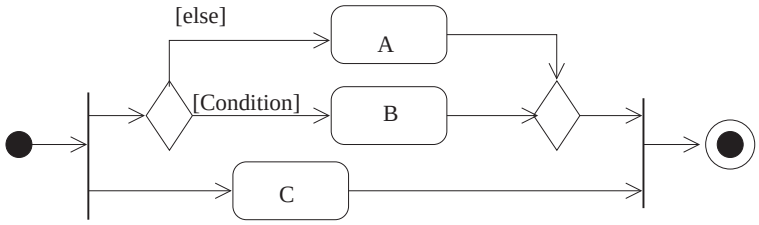

FIG. 13: Exemple d'utilisation des nœuds de contrôles

Nœuds objets et flot de données Nous avons vu que les arcs permettaient de représenter un flot de contrôle. Ce flot de contrôle peut être associé à un flot de données, par exemple pour représenter qu'un action produit une donnée qui sera ensuite utilisée par une autre action (cf. figure 14). Il est possible de représenter le type de la données, le nombre maximum de jetons qu'un nœud objet contient, l'état dans lequel doit se trouver la donnée.

FIG. 14: Nœud objet et flot de données

B.2 Expression des concepts temps réel

Le but de cette section est de fournir des solution permettant de représenter les concepts temps-réel dont vous aurez besoin. Notons que nous ne nous sommes pas restreint aux concepts utilisés dans ce bureau d'étude. On parle de « patrons de conception » (*Design pattern* en anglais). Ces patrons de conception permettent de fournir une solution générique à des problèmes identifiés afin de faciliter le tâche du concepteur d'application.

Représentation des tâches

Une action ou une activité qui possède son propre thread d'exécution est notée avec le mot clé « active ». Nous nous servirons de cette notation pour représenter les tâches de notre application.

Synchronisation de tâches sur données

La figure 21 modélise une tâche périodique qui produit des données et une tâche qui est exécutée lorsque la donnée est produite.

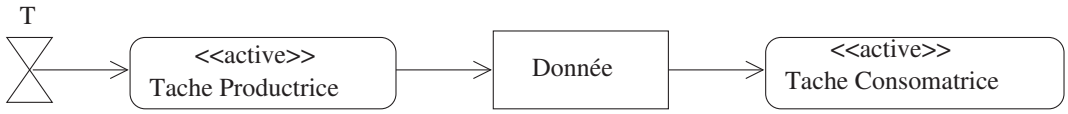

FIG. 15: Synchronisation sur donnée

Événement temporel, tâches périodiques et attente (delay)

La réception d'un événement temporel périodique est représenté au moyen d'un sablier sur lequel est spécifié la période de l'événement (cf. figure 16i)).

La sous-figure ii) représente ainsi l'exécution d'une tâche périodique et le sous figure iii) l'attente d'un temps donnée entre deux actions.

Consommation périodique

La figure 17 modélise une communication inter-tâche par file de messages. La tâche productrice place les données dans un nœud objet dont la taille maximale est spécifiée (100). Ce nœud objet peut par exemple modéliser la présence d'une file ou d'un tableau. Une tâche de période supérieure (100ms) lit l'ensemble des messages contenu dans la file. Le fait que l'ensemble des données est lu en une fois est spécifié par `{weight=all}`.

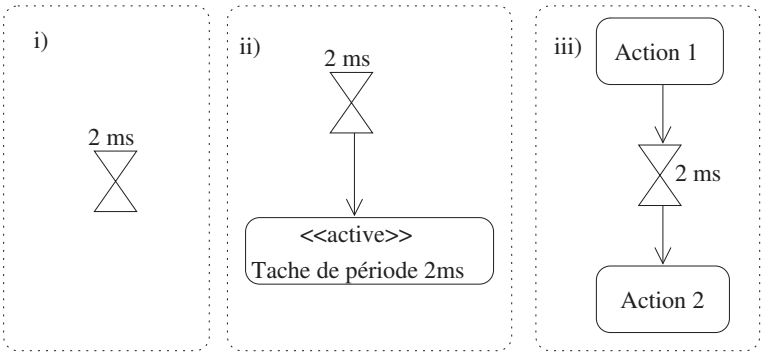

FIG. 16: Événement temporel

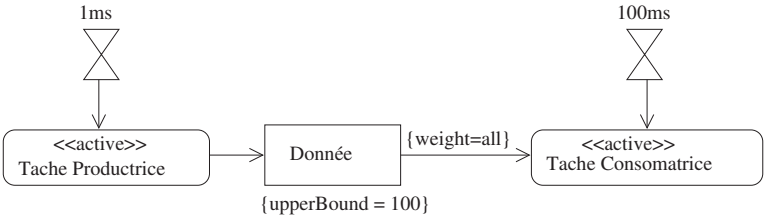

FIG. 17: Consommation périodique

Prise en compte de signaux asynchrone Dans de nombre cas des systèmes temps réel, il est indispensable de prendre en compte des signaux asynchrone, c'est à dire qui peuvent intervenir à tout moment. C'est souvent le cas des signaux issus de l'environnement.

Il faut pouvoir prendre en compte ce type de signaux. Prenons l'exemple, d'un compteur kilométrique d'une voiture. Pour évaluer la distance parcourue par la voiture celui-ci compte le nombre de tour de roue et met en jour une variable entière. L'arrivée d'un tour de roue est un événement asynchrone. La prise en compte d'un tel événement, et la modélisation de la tâche chargée de gérée cet événement sont présentés par la figure 18. Cette figure modélise également l'initialisation à `0` de la variable.

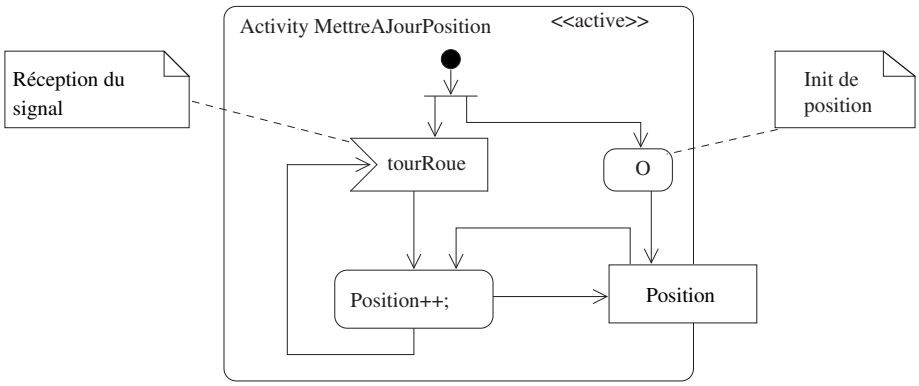

FIG. 18: Prise en compte d'événement asynchrone

[7]

Tâche périodiques avec données d'entrées - variable partagée

Dans de nombreux cas des systèmes temps réel, les tâches communiquent au moyen de variables partagées. Ceci est utilisé lorsque seule la dernière valeur produite est intéressante.

Nous reprenons l'exemple précédent et y ajoutons la tâche `CalculerVitesse` chargée de calculer la vitesse de la voiture à partir de la variable `Position`. L'état de la variable `Position` n'a pas d'influence sur le contrôle (l'exécution) de la tâche `CalculerVitesse`. Or, les nœuds objets contiennent des jetons qui ont une valeur mais qui contrôle également l'exécution des flots descendants. Le modèle de la figure 19 fournit une manière de modéliser une telle relation entre tâche. La tâche `CalculerVitesse` lit la variable position et replace le jeton dans le nœud objet `Position`. Ainsi, dans le cas où aucune position n'est produite entre deux lecture, la dernière valeur de la position est encore disponible (cas où la vitesse est égale à 0). Cette représentation fait également ressortir que `CalculerVitesse` qui utilise la variable ne la modifie pas car aucun arc qui a comme source `CalculerVitesse` et comme cible `Position` n'est présent.

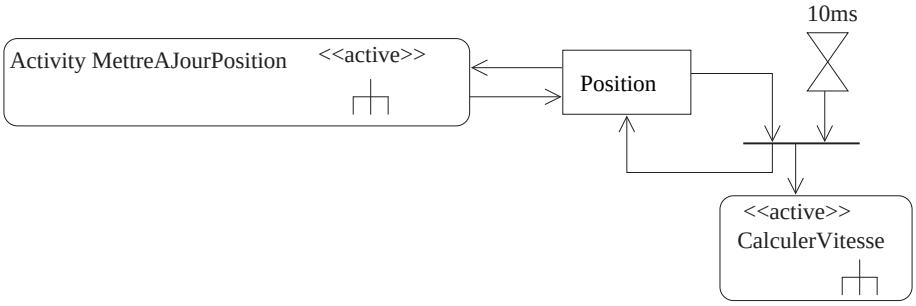

FIG. 19: Variable partagée lue à chaque itération de la tâche

Envoie et réception multiple

Les mots clés « multisend » et « multireceive » permettent de définir des envoie et des réceptions multiples. C'est notamment utile dans le cadre de d'applications client-serveur (cf. figure 20).

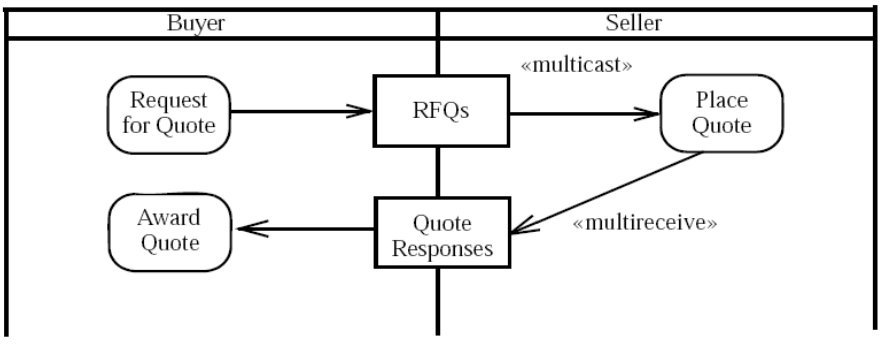

FIG. 20: Envoie et réception multiples

[7] Il est également possible d'exprimer l'envoie de signaux en différentes tâches.

Tâche périodique suspendue sur condition

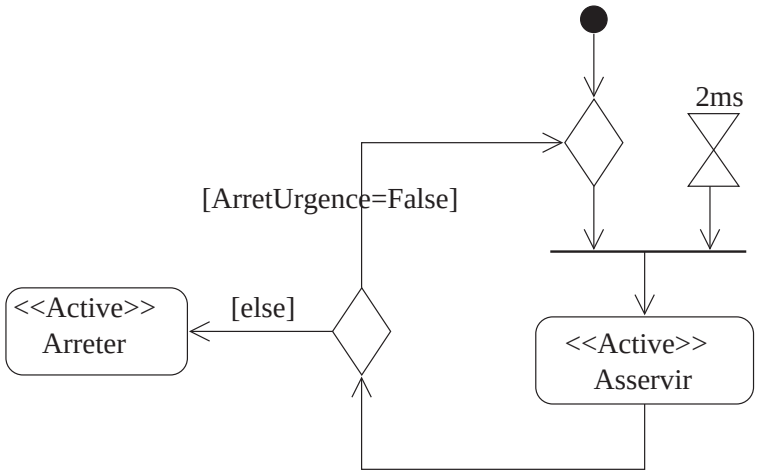

FIG. 21: Exécution d'une tâche et suspension de la tâche sur condition